Table of Contents

TensorFlow-Tutorials 1.1
Simple Multiplication 1.2
Linear Regression 1.3
Logistic Regression 1.4
Feedforward Neural Network (Multilayer Perceptron) 1.5
Deep Feedforward Neural Network (Multilayer Perceptron with 2 Hidden Layers O.o) 1.6
Convolutional Neural Network 1.7
Denoising Autoencoder 1.8
Recurrent Neural Network (LSTM) 1.9
Word2vec 1.10
TensorBoard 1.11
Save and restore net 1.12
Generative Adversarial Network 1.13

TensorFlow-Tutorials

build failing codacy A

> From: nlintz/TensorFlow-Tutorials

Introduction to deep learning based on Google's TensorFlow framework. These tutorials are direct ports of Newmu's Theano Tutorials.

Topics

- Simple Multiplication
- Linear Regression
- Logistic Regression
- Feedforward Neural Network (Multilayer Perceptron)
- Deep Feedforward Neural Network (Multilayer Perceptron with 2 Hidden Layers O.o)
- Convolutional Neural Network
- Denoising Autoencoder
- Recurrent Neural Network (LSTM)
- Word2vec
- TensorBoard
- Save and restore net
- Generative Adversarial Network

Dependencies

- TensorFlow 1.0 alpha
- Numpy
- matplotlib

```
import tensorflow as tf
```

```
a = tf.placeholder("float") # Create a symbolic variable 'a'
b = tf.placeholder("float") # Create a symbolic variable 'b'

y = tf.multiply(a, b) # multiply the symbolic variables
```

```
with tf.Session() as sess: # create a session to evaluate the sy
mbolic expressions
    print("%f should equal 2.0" % sess.run(y, feed_dict={a: 1, b
: 2})) # eval expressions with parameters for a and b
    print("%f should equal 9.0" % sess.run(y, feed_dict={a: 3, b
: 3}))
```

```
2.000000 should equal 2.0
9.000000 should equal 9.0
```

```
import tensorflow as tf
import numpy as np
```

```
trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33 # create a y
value which is approximately linear but with some random noise
```

```
X = tf.placeholder("float") # create symbolic variables
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w) # lr is just X*w so this model line
 is pretty simple

w = tf.Variable(0.0, name="weights") # create a shared variable
(like theano.shared) for the weight matrix
y_model = model(X, w)

cost = tf.square(Y - y_model) # use square error for cost functi
on

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost
) # construct an optimizer to minimize cost and fit line to my d
ata
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize variables (in this case just variab
le W)
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))  # It should be something around 2
```

```
2.00863
```

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

```
def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w):
    return tf.matmul(X, w) # notice we use the same model as lin
ear regression, this is because there is a baked in cost functio
n which performs softmax and cross entropy

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
X = tf.placeholder("float", [None, 784]) # create symbolic varia
bles
Y = tf.placeholder("float", [None, 10])

w = init_weights([784, 10]) # like in linear regression, we need
 a shared variable weight matrix for logistic regression

py_x = model(X, w)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y)) # compute mean cross entropy (softmax is a
pplied internally)
train_op = tf.train.GradientDescentOptimizer(0.05).minimize(cost
) # construct optimizer
predict_op = tf.argmax(py_x, 1) # at predict time, evaluate the
argmax of the logistic regression
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end]})
        print(i, np.mean(np.argmax(teY, axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX}
)))
```

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

```
def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w_h, w_o):
    h = tf.nn.sigmoid(tf.matmul(X, w_h)) # this is a basic mlp,
think 2 stacked logistic regressions
    return tf.matmul(h, w_o) # note that we dont take the softma
x at the end because our cost fn does that for us

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
X = tf.placeholder("float", [None, 784])
Y = tf.placeholder("float", [None, 10])

w_h = init_weights([784, 625]) # create symbolic variables
w_o = init_weights([625, 10])

py_x = model(X, w_h, w_o)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y)) # compute costs
train_op = tf.train.GradientDescentOptimizer(0.05).minimize(cost
) # construct an optimizer
predict_op = tf.argmax(py_x, 1)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end]})
        print(i, np.mean(np.argmax(teY, axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX}
)))
```

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

```
def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden): # thi
s network is the same as the previous one except with an extra h
idden layer + dropout
    X = tf.nn.dropout(X, p_keep_input)
    h = tf.nn.relu(tf.matmul(X, w_h))

    h = tf.nn.dropout(h, p_keep_hidden)
    h2 = tf.nn.relu(tf.matmul(h, w_h2))

    h2 = tf.nn.dropout(h2, p_keep_hidden)

    return tf.matmul(h2, w_o)

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
X = tf.placeholder("float", [None, 784])
Y = tf.placeholder("float", [None, 10])

w_h = init_weights([784, 625])
w_h2 = init_weights([625, 625])
w_o = init_weights([625, 10])

p_keep_input = tf.placeholder("float")
p_keep_hidden = tf.placeholder("float")
py_x = model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y))
train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
predict_op = tf.argmax(py_x, 1)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end],
                                          p_keep_input: 0.8, p_k
eep_hidden: 0.5})
        print(i, np.mean(np.argmax(teY, axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX,
 Y: teY,
                                                         p_keep_
input: 1.0,
                                                         p_keep_
hidden: 1.0})))
```

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

```
batch_size = 128
test_size = 256

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w, w2, w3, w4, w_o, p_keep_conv, p_keep_hidden):
    l1a = tf.nn.relu(tf.nn.conv2d(X, w,                       #
l1a shape=(?, 28, 28, 32)
                        strides=[1, 1, 1, 1], padding='SAME'))
    l1 = tf.nn.max_pool(l1a, ksize=[1, 2, 2, 1],              #
l1 shape=(?, 14, 14, 32)
                        strides=[1, 2, 2, 1], padding='SAME')
    l1 = tf.nn.dropout(l1, p_keep_conv)

    l2a = tf.nn.relu(tf.nn.conv2d(l1, w2,                     #
l2a shape=(?, 14, 14, 64)
                        strides=[1, 1, 1, 1], padding='SAME'))
    l2 = tf.nn.max_pool(l2a, ksize=[1, 2, 2, 1],              #
l2 shape=(?, 7, 7, 64)
                        strides=[1, 2, 2, 1], padding='SAME')
    l2 = tf.nn.dropout(l2, p_keep_conv)

    l3a = tf.nn.relu(tf.nn.conv2d(l2, w3,                     #
l3a shape=(?, 7, 7, 128)
                        strides=[1, 1, 1, 1], padding='SAME'))
    l3 = tf.nn.max_pool(l3a, ksize=[1, 2, 2, 1],              #
l3 shape=(?, 4, 4, 128)
                        strides=[1, 2, 2, 1], padding='SAME')
    l3 = tf.reshape(l3, [-1, w4.get_shape().as_list()[0]])    #
reshape to (?, 2048)
    l3 = tf.nn.dropout(l3, p_keep_conv)

    l4 = tf.nn.relu(tf.matmul(l3, w4))
    l4 = tf.nn.dropout(l4, p_keep_hidden)

    pyx = tf.matmul(l4, w_o)
    return pyx

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
trX = trX.reshape(-1, 28, 28, 1)  # 28x28x1 input img
teX = teX.reshape(-1, 28, 28, 1)  # 28x28x1 input img
```

```
X = tf.placeholder("float", [None, 28, 28, 1])
Y = tf.placeholder("float", [None, 10])

w = init_weights([3, 3, 1, 32])       # 3x3x1 conv, 32 outputs
w2 = init_weights([3, 3, 32, 64])     # 3x3x32 conv, 64 outputs
w3 = init_weights([3, 3, 64, 128])    # 3x3x32 conv, 128 outputs
w4 = init_weights([128 * 4 * 4, 625]) # FC 128 * 4 * 4 inputs, 6
25 outputs
w_o = init_weights([625, 10])         # FC 625 inputs, 10 output
s (labels)

p_keep_conv = tf.placeholder("float")
p_keep_hidden = tf.placeholder("float")
py_x = model(X, w, w2, w3, w4, w_o, p_keep_conv, p_keep_hidden)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y))
train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
predict_op = tf.argmax(py_x, 1)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        training_batch = zip(range(0, len(trX), batch_size),
                             range(batch_size, len(trX)+1, batch
_size))
        for start, end in training_batch:
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end],
                                          p_keep_conv: 0.8, p_ke
ep_hidden: 0.5})

        test_indices = np.arange(len(teX)) # Get A Test Batch
        np.random.shuffle(test_indices)
        test_indices = test_indices[0:test_size]

        print(i, np.mean(np.argmax(teY[test_indices], axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX[
test_indices],
                                                         Y: teY[
test_indices],
                                                         p_keep_
conv: 1.0,
                                                         p_keep_
hidden: 1.0})))
```

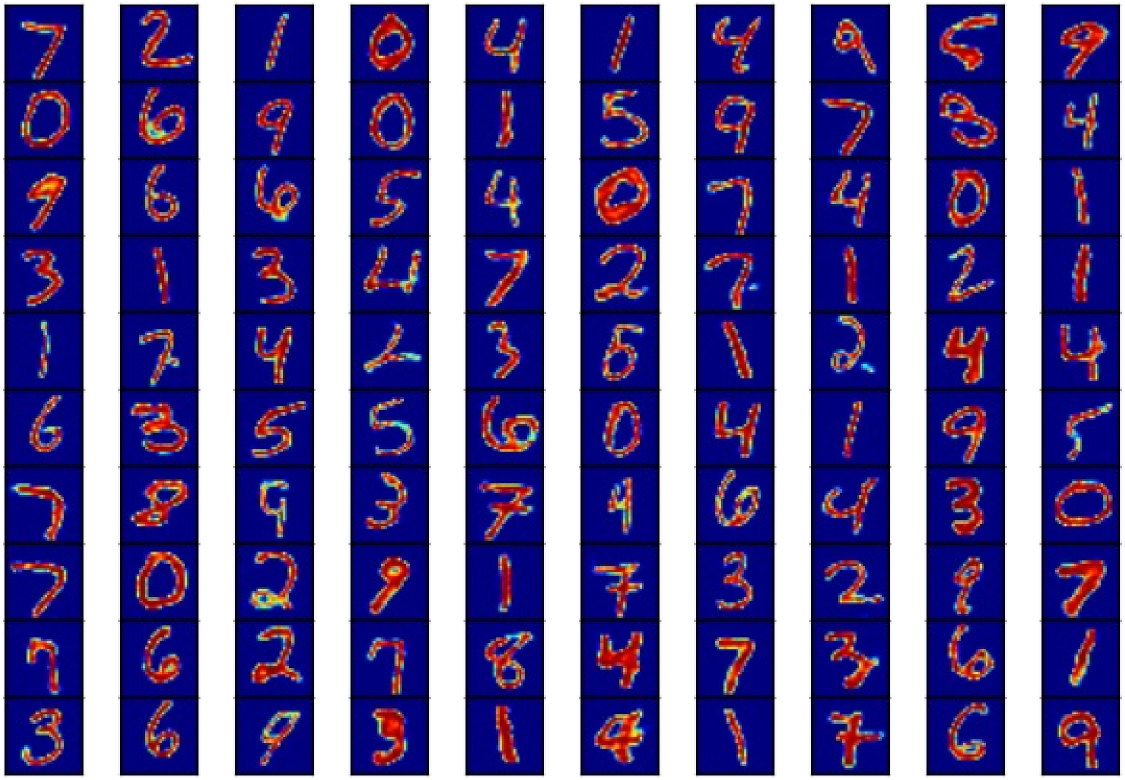
```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
%matplotlib inline
```

```
## Visualizing reconstructions
def vis(images, save_name):
    dim = images.shape[0]
    n_image_rows = int(np.ceil(np.sqrt(dim)))
    n_image_cols = int(np.ceil(dim * 1.0/n_image_rows))
    gs = gridspec.GridSpec(n_image_rows,n_image_cols,top=1., bot
tom=0., right=1., left=0., hspace=0., wspace=0.)
    for g,count in zip(gs,range(int(dim))):
        ax = plt.subplot(g)
        ax.imshow(images[count,:].reshape((28,28)))
        ax.set_xticks([])
        ax.set_yticks([])
    plt.savefig(save_name + '_vis.png')
    plt.show()
```

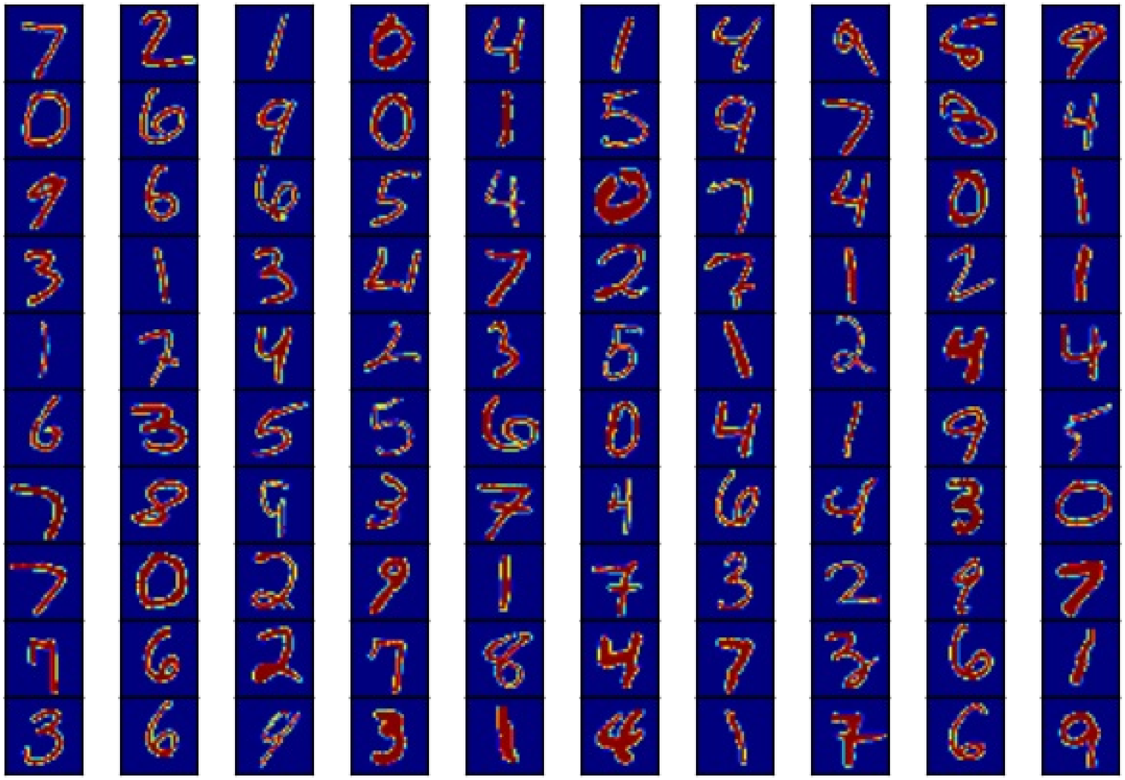
```
mnist_width = 28
n_visible = mnist_width * mnist_width
n_hidden = 500
corruption_level = 0.3

# create node for input data
X = tf.placeholder("float", [None, n_visible], name='X')

# create node for corruption mask
mask = tf.placeholder("float", [None, n_visible], name='mask')

# create nodes for hidden variables
W_init_max = 4 * np.sqrt(6. / (n_visible + n_hidden))
W_init = tf.random_uniform(shape=[n_visible, n_hidden],
                           minval=-W_init_max,
                           maxval=W_init_max)

W = tf.Variable(W_init, name='W')
b = tf.Variable(tf.zeros([n_hidden]), name='b')

W_prime = tf.transpose(W)  # tied weights between encoder and de
coder
b_prime = tf.Variable(tf.zeros([n_visible]), name='b_prime')


def model(X, mask, W, b, W_prime, b_prime):
    tilde_X = mask * X  # corrupted X

    Y = tf.nn.sigmoid(tf.matmul(tilde_X, W) + b)  # hidden state
    Z = tf.nn.sigmoid(tf.matmul(Y, W_prime) + b_prime)  # recons
tructed input
    return Z

# build model graph
Z = model(X, mask, W, b, W_prime, b_prime)

# create cost function
cost = tf.reduce_sum(tf.pow(X - Z, 2))  # minimize squared error
train_op = tf.train.GradientDescentOptimizer(0.02).minimize(cost
)  # construct an optimizer
predict_op = Z
# load MNIST data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            input_ = trX[start:end]
            mask_np = np.random.binomial(1, 1 - corruption_level
, input_.shape)
            sess.run(train_op, feed_dict={X: input_, mask: mask_
np})

        mask_np = np.random.binomial(1, 1 - corruption_level, te
X.shape)
        print(i, sess.run(cost, feed_dict={X: teX, mask: mask_np
}))
    # save the predictions for 100 images
    mask_np = np.random.binomial(1, 1 - corruption_level, teX[:1
00].shape)
    predicted_imgs = sess.run(predict_op, feed_dict={X: teX[:100
], mask: mask_np})
    input_imgs = teX[:100]
```

```
# Plot the reconstructed images
vis(predicted_imgs,'pred')
```



```
# Plot input images to compare with
vis(input_imgs,'in')
```



```
#Inspired by https://github.com/aymericdamien/TensorFlow-Example
s/blob/master/examples/3%20-%20Neural%20Networks/recurrent_netwo
rk.py
import tensorflow as tf
import numpy as np
from tensorflow.contrib import rnn
from tensorflow.examples.tutorials.mnist import input_data
```

```
# configuration
#                         O * W + b -> 10 labels for each image,
O[? 28], W[28 10], B[10]
#                        ^ (O: output 28 vec from 28 vec input)
#                        |
#      +-+  +-+       +--+
#      |1|->|2|-> ... |28| time_step_size = 28
#      +-+  +-+       +--+
#       ^    ^    ...  ^
#       |    |         |
# img1:[28] [28]  ... [28]
# img2:[28] [28]  ... [28]
# img3:[28] [28]  ... [28]
# ...
# img128 or img256 (batch_size or test_size 256)
#      each input size = input_vec_size=lstm_size=28

# configuration variables
input_vec_size = lstm_size = 28
time_step_size = 28

batch_size = 128
test_size = 256

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, W, B, lstm_size):
    # X, input shape: (batch_size, time_step_size, input_vec_siz
e)
    XT = tf.transpose(X, [1, 0, 2])  # permute time_step_size an
d batch_size
    # XT shape: (time_step_size, batch_size, input_vec_size)
    XR = tf.reshape(XT, [-1, lstm_size]) # each row has input fo
r each lstm cell (lstm_size=input_vec_size)
    # XR shape: (time_step_size * batch_size, input_vec_size)
    X_split = tf.split(XR, time_step_size, 0) # split them to ti
me_step_size (28 arrays)
    # Each array shape: (batch_size, input_vec_size)

    # Make lstm with lstm_size (each input vector size)
    lstm = rnn.BasicLSTMCell(lstm_size, forget_bias=1.0, state_i
```

```
s_tuple=True)

    # Get lstm cell output, time_step_size (28) arrays with lstm
_size output: (batch_size, lstm_size)
    outputs, _states = rnn.static_rnn(lstm, X_split, dtype=tf.fl
oat32)

    # Linear activation
    # Get the last output
    return tf.matmul(outputs[-1], W) + B, lstm.state_size # Stat
e size to initialize the stat

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
trX = trX.reshape(-1, 28, 28)
teX = teX.reshape(-1, 28, 28)
```

```
X = tf.placeholder("float", [None, 28, 28])
Y = tf.placeholder("float", [None, 10])

# get lstm_size and output 10 labels
W = init_weights([lstm_size, 10])
B = init_weights([10])

py_x, state_size = model(X, W, B, lstm_size)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y))
train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
predict_op = tf.argmax(py_x, 1)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), batch_size), ra
nge(batch_size, len(trX)+1, batch_size)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end]})

        test_indices = np.arange(len(teX))  # Get A Test Batch
        np.random.shuffle(test_indices)
        test_indices = test_indices[0:test_size]

        print(i, np.mean(np.argmax(teY[test_indices], axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX[
test_indices]})))
```

```
# Inspired by https://www.tensorflow.org/versions/r0.7/tutorials
/word2vec/index.html
import collections
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

```
# Configuration
batch_size = 20
# Dimension of the embedding vector. Two too small to get
# any meaningful embeddings, but let's make it 2 for simple visu
alization
embedding_size = 2
num_sampled = 15    # Number of negative examples to sample.

# Sample sentences
sentences = ["the quick brown fox jumped over the lazy dog",
            "I love cats and dogs",
            "we all love cats and dogs",
            "cats and dogs are great",
            "sung likes cats",
            "she loves dogs",
            "cats can be very independent",
            "cats are great companions when they want to be",
            "cats are playful",
            "cats are natural hunters",
            "It's raining cats and dogs",
            "dogs and cats love sung"]

# sentences to words and count
words = " ".join(sentences).split()
count = collections.Counter(words).most_common()
print ("Word count", count[:5])

# Build dictionaries
rdic = [i[0] for i in count] #reverse dic, idx -> word
dic = {w: i for i, w in enumerate(rdic)} #dic, word -> id
voc_size = len(dic)

# Make indexed word data
data = [dic[word] for word in words]
print('Sample data', data[:10], [rdic[t] for t in data[:10]])

# Let's make a training data for window size 1 for simplicity
# ([the, brown], quick), ([quick, fox], brown), ([brown, jumped]
, fox), ...
cbow_pairs = [];
for i in range(1, len(data)-1) :
    cbow_pairs.append([[data[i-1], data[i+1]], data[i]]);
```

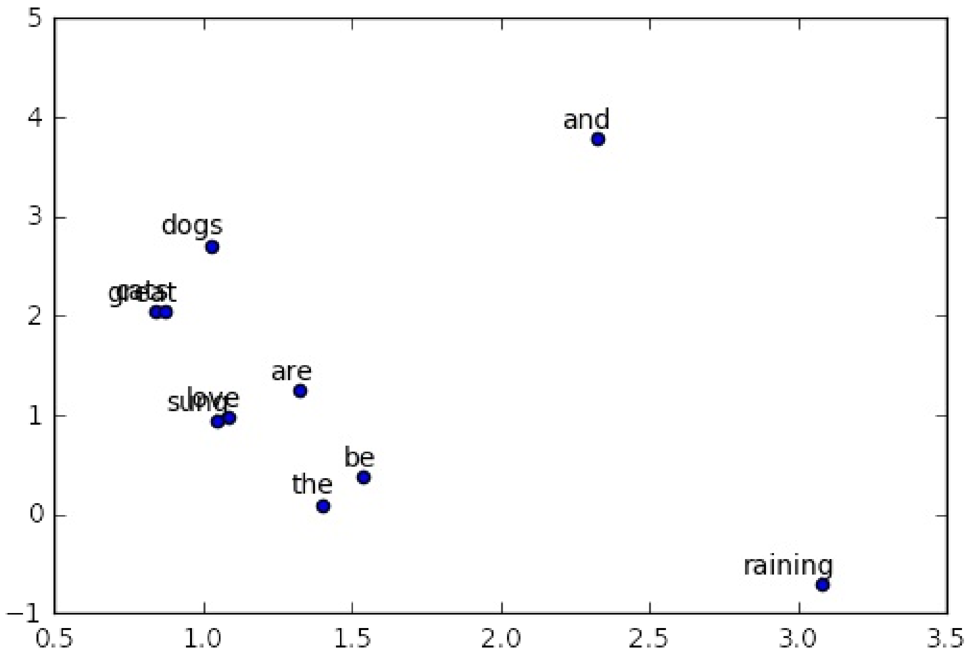
```
print('Context pairs', cbow_pairs[:10])

# Let's make skip-gram pairs
# (quick, the), (quick, brown), (brown, quick), (brown, fox), ...

skip_gram_pairs = [];
for c in cbow_pairs:
    skip_gram_pairs.append([c[1], c[0][0]])
    skip_gram_pairs.append([c[1], c[0][1]])
print('skip-gram pairs', skip_gram_pairs[:5])

def generate_batch(size):
    assert size < len(skip_gram_pairs)
    x_data=[]
    y_data = []
    r = np.random.choice(range(len(skip_gram_pairs)), size, repl
ace=False)
    for i in r:
        x_data.append(skip_gram_pairs[i][0])  # n dim
        y_data.append([skip_gram_pairs[i][1]])  # n, 1 dim
    return x_data, y_data

# generate_batch test
print ('Batches (x, y)', generate_batch(3))
```

```
('Word count', [('cats', 10), ('dogs', 6), ('and', 5), ('are', 4
), ('love', 3)])
('Sample data', [8, 33, 24, 20, 17, 12, 8, 25, 30, 26], ['the',
'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog',
 'I'])
('Context pairs', [[[8, 24], 33], [[33, 20], 24], [[24, 17], 20]
, [[20, 12], 17], [[17, 8], 12], [[12, 25], 8], [[8, 30], 25], [
[25, 26], 30], [[30, 4], 26], [[26, 0], 4]])
('skip-gram pairs', [[33, 8], [33, 24], [24, 33], [24, 20], [20,
 24]])
('Batches (x, y)', ([27, 15, 22], [[0], [28], [18]]))
```

```
# Input data
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
# need to shape [batch_size, 1] for nn.nce_loss
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
# Ops and variables pinned to the CPU because of missing GPU imp
lementation
with tf.device('/cpu:0'):
    # Look up embeddings for inputs.
    embeddings = tf.Variable(
        tf.random_uniform([voc_size, embedding_size], -1.0, 1.0)
)
    embed = tf.nn.embedding_lookup(embeddings, train_inputs) # l
ookup table

# Construct the variables for the NCE loss
nce_weights = tf.Variable(
    tf.random_uniform([voc_size, embedding_size],-1.0, 1.0))
nce_biases = tf.Variable(tf.zeros([voc_size]))

# Compute the average NCE loss for the batch.
# This does the magic:
#   tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled,
 num_classes ...)
# It automatically draws negative samples when we evaluate the l
oss.
loss = tf.reduce_mean(tf.nn.nce_loss(nce_weights, nce_biases, tr
ain_labels, embed, num_sampled, voc_size))
# Use the adam optimizer
train_op = tf.train.AdamOptimizer(1e-1).minimize(loss)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # Initializing all variables
    tf.global_variables_initializer().run()

    for step in range(100):
        batch_inputs, batch_labels = generate_batch(batch_size)
        _, loss_val = sess.run([train_op, loss],
                feed_dict={train_inputs: batch_inputs, train_lab
els: batch_labels})
        if step % 10 == 0:
          print("Loss at ", step, loss_val) # Report the loss

    # Final embeddings are ready for you to use. Need to normali
ze for practical use
    trained_embeddings = embeddings.eval()

# Show word2vec if dim is 2
if trained_embeddings.shape[1] == 2:
    labels = rdic[:10] # Show top 10 words
    for i, label in enumerate(labels):
        x, y = trained_embeddings[i,:]
        plt.scatter(x, y)
        plt.annotate(label, xy=(x, y), xytext=(5, 2),
            textcoords='offset points', ha='right', va='bottom')
    plt.savefig("word2vec.png")
```

```
('Loss at ', 0, 16.654182)
('Loss at ', 10, 14.677063)
('Loss at ', 20, 9.1576614)
('Loss at ', 30, 3.9546738)
('Loss at ', 40, 3.8289108)
('Loss at ', 50, 3.3630223)
('Loss at ', 60, 3.6222715)
('Loss at ', 70, 3.1979971)
('Loss at ', 80, 3.5327618)
('Loss at ', 90, 3.4316573)
```


and
dogs
great
cats
are
love
sung
be
the
raining
5
4
3
2
1
0
−1
0.5
1.0
1.5
2.0
2.5
3.0
3.5

TensorBoard

After Training the model, run

```
tensorboard --logdir=path/to/log-directory
```

Tensorboard provides a good visualization tool for all the variables you like and works on a browser.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
def init_weights(shape, name):
    return tf.Variable(tf.random_normal(shape, stddev=0.01), nam
e=name)

# This network is the same as the previous one except with an ex
tra hidden layer + dropout
def model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden):
    # Add layer name scopes for better graph visualization
    with tf.name_scope("layer1"):
        X = tf.nn.dropout(X, p_keep_input)
        h = tf.nn.relu(tf.matmul(X, w_h))
    with tf.name_scope("layer2"):
        h = tf.nn.dropout(h, p_keep_hidden)
        h2 = tf.nn.relu(tf.matmul(h, w_h2))
    with tf.name_scope("layer3"):
        h2 = tf.nn.dropout(h2, p_keep_hidden)
        return tf.matmul(h2, w_o)

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
X = tf.placeholder("float", [None, 784], name="X")
Y = tf.placeholder("float", [None, 10], name="Y")

w_h = init_weights([784, 625], "w_h")
w_h2 = init_weights([625, 625], "w_h2")
w_o = init_weights([625, 10], "w_o")

# Add histogram summaries for weights
tf.summary.histogram("w_h_summ", w_h)
tf.summary.histogram("w_h2_summ", w_h2)
tf.summary.histogram("w_o_summ", w_o)

p_keep_input = tf.placeholder("float", name="p_keep_input")
p_keep_hidden = tf.placeholder("float", name="p_keep_hidden")
py_x = model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden)

with tf.name_scope("cost"):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logit
s(logits=py_x, labels=Y))
    train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(co
st)
    # Add scalar summary for cost
    tf.summary.scalar("cost", cost)

with tf.name_scope("accuracy"):
    correct_pred = tf.equal(tf.argmax(Y, 1), tf.argmax(py_x, 1))
# Count correct predictions
    acc_op = tf.reduce_mean(tf.cast(correct_pred, "float")) # Ca
st boolean to float to average
    # Add scalar summary for accuracy
    tf.summary.scalar("accuracy", acc_op)
```

```
with tf.Session() as sess:
    # create a log writer. run 'tensorboard --logdir=./logs/nn_l
ogs'
    writer = tf.summary.FileWriter("./logs/nn_logs", sess.graph)
# for 1.0
    merged = tf.summary.merge_all()

    # you need to initialize all variables
    tf.global_variables_initializer().run()

    for i in range(100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end],
                                          p_keep_input: 0.8, p_k
eep_hidden: 0.5})
        summary, acc = sess.run([merged, acc_op], feed_dict={X:
teX, Y: teY,
                                          p_keep_input: 1.0, p_k
eep_hidden: 1.0})
        writer.add_summary(summary, i)  # Write summary
        print(i, acc)                   # Report the accuracy
```

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
import os
```

```
# This shows how to save/restore your model (trained variables).
# To see how it works, please stop this program during training
and resart.
# This network is the same as 3_net.py

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden): # thi
s network is the same as the previous one except with an extra h
idden layer + dropout
    X = tf.nn.dropout(X, p_keep_input)
    h = tf.nn.relu(tf.matmul(X, w_h))

    h = tf.nn.dropout(h, p_keep_hidden)
    h2 = tf.nn.relu(tf.matmul(h, w_h2))

    h2 = tf.nn.dropout(h2, p_keep_hidden)

    return tf.matmul(h2, w_o)

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
trX, trY, teX, teY = mnist.train.images, mnist.train.labels, mni
st.test.images, mnist.test.labels
```

```
X = tf.placeholder("float", [None, 784])
Y = tf.placeholder("float", [None, 10])

w_h = init_weights([784, 625])
w_h2 = init_weights([625, 625])
w_o = init_weights([625, 10])

p_keep_input = tf.placeholder("float")
p_keep_hidden = tf.placeholder("float")
py_x = model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden)

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=py_x, labels=Y))
train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
predict_op = tf.argmax(py_x, 1)

ckpt_dir = "./ckpt_dir"
if not os.path.exists(ckpt_dir):
    os.makedirs(ckpt_dir)

global_step = tf.Variable(0, name='global_step', trainable=False
)

# Call this after declaring all tf.Variables.
saver = tf.train.Saver()

# This variable won't be stored, since it is declared after tf.t
rain.Saver()
non_storable_variable = tf.Variable(777)
```

```
# Launch the graph in a session
with tf.Session() as sess:
    # you need to initialize all variables
    tf.global_variables_initializer().run()

    ckpt = tf.train.get_checkpoint_state(ckpt_dir)
    if ckpt and ckpt.model_checkpoint_path:
        print(ckpt.model_checkpoint_path)
        saver.restore(sess, ckpt.model_checkpoint_path) # restor
e all variables

    start = global_step.eval() # get last global_step
    print("Start from:", start)

    for i in range(start, 100):
        for start, end in zip(range(0, len(trX), 128), range(128
, len(trX)+1, 128)):
            sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end],
                                          p_keep_input: 0.8, p_k
eep_hidden: 0.5})

        global_step.assign(i).eval() # set and update(eval) glob
al_step with index, i
        saver.save(sess, ckpt_dir + "/model.ckpt", global_step=g
lobal_step)
        print(i, np.mean(np.argmax(teY, axis=1) ==
                         sess.run(predict_op, feed_dict={X: teX,
 Y: teY,
                                                         p_keep_
input: 1.0,
                                                         p_keep_
hidden: 1.0})))
```

Generative Adversarial Networks

This notebook implements a very basic GAN with MLPs for the 2 networks.

```
%matplotlib inline

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets("MNIST_data/")
images = mnist.train.images
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

Weight initialisation

The weights will be initiliased using the Xavier initialisation method [1]. In this case, this is just a Gaussian distribution with a custom standard deviation: the standard deviation is inversely proportional to the number of neurons feeding into the neuron.

$$ \mathbf{w}_i \sim \mathcal{N}(0, \frac{1}{n_{i-1}}) $$

where n_{i-1} is the number of inputs that feed into the current neuron.

I also tried with regular Gaussian (i.e. constant σ) and with uniform distribution, but I did not manage to get the network learning.

[1] *Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Aistats. Vol. 9. 2010.*

```
def xavier_initializer(shape):
    return tf.random_normal(shape=shape, stddev=1/shape[0])
```

Architecture

```
# Generator
z_size = 100 # Latent vector dimension
g_w1_size = 400
g_out_size = 28 * 28

# Discriminator
x_size = 28 * 28
d_w1_size = 400
d_out_size = 1
```

```
z = tf.placeholder('float', shape=(None, z_size))
X = tf.placeholder('float', shape=(None, x_size))
```

Weights

```
g_weights = {
    'w1': tf.Variable(xavier_initializer(shape=(z_size, g_w1_siz
e))),
    'b1': tf.Variable(tf.zeros(shape=[g_w1_size])),
    'out': tf.Variable(xavier_initializer(shape=(g_w1_size, g_ou
t_size))),
    'b2': tf.Variable(tf.zeros(shape=[g_out_size])),
}

d_weights ={
    'w1': tf.Variable(xavier_initializer(shape=(x_size, d_w1_siz
e))),
    'b1': tf.Variable(tf.zeros(shape=[d_w1_size])),
    'out': tf.Variable(xavier_initializer(shape=(d_w1_size, d_ou
t_size))),
    'b2': tf.Variable(tf.zeros(shape=[d_out_size])),
}
```

Models

The models were chosen to be very simple, so just an MLP with 1 hidden layer and 1 output layer.

```
def G(z, w=g_weights):
    h1 = tf.nn.relu(tf.matmul(z, w['w1']) + w['b1'])
    return tf.sigmoid(tf.matmul(h1, w['out']) + w['b2'])

def D(x, w=d_weights):
    h1 = tf.nn.relu(tf.matmul(x, w['w1']) + w['b1'])
    return tf.sigmoid(tf.matmul(h1, w['out']) + w['b2'])
```

Latent distribution

This function generates a prior for G.

```
def generate_z(n=1):
    return np.random.normal(size=(n, z_size))
```

```
sample = G(z) # To be called during session
```

Cost

The cost functions are the ones used in the original GAN paper [2], using the suggestion of switching the loss for G from minimising $\frac{1}{m}\sum*{i=1}^m(1-D(G(\mathbf{z}))$ to maximising $\frac{1}{m}\sum*{i=1}^m(D(G(\mathbf{z}))$.

Note that because both need to be maximised, and TF is designed to minimise, we take the negative values below.

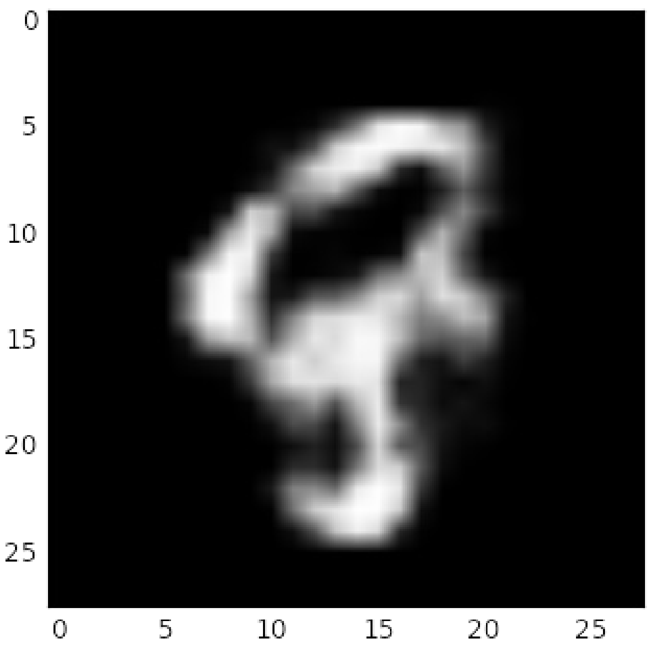
[2] *Goodfellow, Ian, et al. "Generative adversarial nets." Advances in neural information processing systems. 2014.*

```
G_objective = -tf.reduce_mean(tf.log(D(G(z))))
D_objective = -tf.reduce_mean(tf.log(D(X)) + tf.log(1 - D(G(z)))
)
```

Optimisation

Note that each of the optimiser takes a `var_list` argument to only consider the variables provided. This is because we don't want D to train G when D is trained, but rather freeze the weights from G and only concern about D (and the same for G).

```
G_opt = tf.train.AdamOptimizer().minimize(
    G_objective, var_list=g_weights.values())
D_opt = tf.train.AdamOptimizer().minimize(
    D_objective, var_list=d_weights.values())
```

Training

```
# Hyper-parameters
epochs = 50000
batch_size = 128

# Session
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for _ in range(epochs):
        sess.run(G_opt, feed_dict={
            z: generate_z(batch_size)
        })
        sess.run(D_opt, feed_dict={
            X: images[np.random.choice(range(len(images)), batch
_size)].reshape(batch_size, x_size),
            z: generate_z(batch_size),
        })

    # Show a random image
    image = sess.run(sample, feed_dict={z:generate_z()})
    plt.imshow(image.reshape(28, 28), cmap='gray')
```